Interfacing ACT-R 5.0 to an Unmanned Aerial Vehicle (UAV) Synthetic Task Environment (STE)

Jerry T. Ball
Kevin A. Gluck
(jerry.ball@mesa.afmc.af.mil; kevin.gluck@mesa.afmc.af.mil)

With what external system have you attached ACT-R?

We have attached ACT-R to the Air Force Research Laboratory's (AFRL) Unmanned Aerial Vehicle (UAV) Synthetic Task Environment (STE). The UAV STE is a high fidelity simulation of the Predator RQ1A Unmanned Aerial Vehicle with built in basic maneuvering, landing and reconnaissance tasks and data collection capabilities. The UAV STE's design and capabilities are described in detail in Schreiber, Lyon, Martin, and Confer (2002). We are currently finishing up the modeling of the basic maneuvering task and have begun the modeling of the reconnaissance task. Details about the ACT-R model of basic maneuvering can be found in Gluck, Ball, Krusmark, Rodgers, and Purtee (2003) and in Ball, Gluck, Krusmark, and Rodgers (2003).

The UAV STE is a complex software system—coded in C—which is composed of multiple processes running on Intel-based hardware under the Windows 2000 operating system. There are a variety of processes running simultaneously when the STE is in use. These processes cluster into two groups. One group is the Instructor Operator Station (IOS). The IOS processes allow the user (typically the experimenter) to control the parameters of the basic maneuvering, landing, and reconnaissance tasks. The other group of processes comprise the Pilot Station. This includes the control inputs process, aerodynamics module, graphics display process, and others, which allow the user to fly the simulation. All of these processes communicate with each other using datagram sockets with pre-determined port numbers and shared memory—passing back and forth a collection of data structures that capture information about the current state of the system.

Communication between processes is mediated by a "heartbeat" mechanism under the control of a separate "OS" process that runs on each hardware platform. This heartbeat mechanism serves to synchronize the processes by sending events to the process loop of each process at heartbeat intervals of approximately 20 msec. When a process, triggered by a heartbeat event, reads data from a datagram socket, it does so in a loop, throwing out all but the last datagram in the socket buffer. This ensures that the receiving process retrieves the latest data from the sending process. This socket "polling" mechanism requires the use of non-blocking sockets, since it is failure to retrieve a datagram from the socket buffer that triggers termination of the loop and this would not happen if the read blocked waiting for a datagram to arrive. The heartbeat mechanism also triggers the sending of data from one process to another. The Allegro Common Lisp (ACL) implementation of sockets does not support non-blocking sockets.

How was it done? How is communication over the network handled? What information is communicated back and forth?

The external interface to the UAV STE relies on use of the communications and synchronization mechanisms described above. These mechanisms are proprietary and poorly documented. Although not documented, it is possible to introduce a new process that hooks into these existing mechanisms without modifying the existing code. It is also possible to modify the source code for several of the processes (where the source code is available) to meet new requirements. However, there is no access to the source code for the core OS process and the heartbeat mechanism.

The cognitive model runs on a separate computer, also under Windows 2000. The cognitive model runs in ACT-R 5.0, on top of Allegro Common Lisp (ACL) 6.2. Separate mechanisms are required to send input to the UAV STE and to receive outputs from the UAV STE. To send input to the UAV STE, the cognitive model contains supporting Lisp code that communicates with a modified version of the control inputs process. The control inputs process was modified to take control inputs from a datagram socket rather than the controls themselves. The lisp code in the cognitive model sends datagrams containing the control inputs to the socket that the modified control inputs process is listening on. The sending of datagrams to the modified control inputs process relies on the normal Windows event processing mechanism for signaling receipt of a datagram on a socket and does not rely on the use of the proprietary heartbeat and datagram socket polling mechanism. However, the sending of control inputs data from the cognitive model occurs infrequently enough (1 to 2 times per second) to circumvent the heartbeat and socket polling mechanism. The modified control inputs process runs on the cognitive model computer rather than the UAV STE computer. It makes the control inputs available to the rest of the UAV STE processes using the proprietary heartbeat mechanism and reuses the previous values until new values are received from the cognitive model. To receive outputs from the UAV STE a modified version of the core OS process was created by the developer of the UAV STE. This modified OS process runs on the cognitive model computer and communicates with the UAV STE via the heartbeat mechanism and socket polling. At each heartbeat, it receives the Variable Information Table (VIT) data structure from the UAV STE. The VIT contains 100's of variables describing the current state of the UAV STE including the current value of indicators in the Heads Up Display (HUD) of the UAV STE (e.g. airspeed, altitude, heading, vertical speed). The modified OS process (written in C) has a foreign function interface to the Lisp code of the cognitive model and makes the values of relevant variables available to the cognitive model. The foreign function interface requires inclusion of a DLL into the Lisp process. This DLL exposes a foreign function to the Lisp code which is used to retrieve the values of state variables from the VIT.

Once the state data are available to Lisp, they are used to update a Lisp-based mock-up of the STE's Heads-Up Display (HUD). This is what the ACT-R model actually "looks" at when it is flying the simulator, because it can't encode data directly off the C-based HUD. Reimplementation of the HUD in lisp was necessary to be able to use the RPM component of ACT-R. However, there was never any consideration of reimplementing the entire UAV STE in lisp and the goal was always to minimize the amount of recoding that was required. The RPM component of ACT-R comes with a limited set of object types built in (e.g. static text, button, line). These built in objects have added functionality (over simple graphic objects) to support the perceptual component of ACT-R. The RPM component of ACT-R is implemented using the object oriented capabilities of Common Lisp and RPM provides a set of classes for building new types of graphical objects which incorporate this added functionality. The documentation on how to use these classes to build new types of objects is limited and to some extent specific to the MAC Lisp implementation of RPM. Creating new objects to represent the horizon line and reticle of the UAV HUD required working around these issues. Support from Dan Bothell was critical to our success in doing this. The classes for building new graphical objects allow those objects to encode non-perceptual information (i.e. information that is not retrievable from the visual form of the object). We currently use this capability to provide digital values for the bank angle and pitch in the horizon line/reticle object. This avoids the need to do any lower level visual processing of the horizon line and reticle to determine these values based on geometric position, but it provides a discrete, quantitative read-out value for what is presumably an analog and qualitative process and in so doing it pushes higher level cognitive processing capabilities into what is a lower level perceptual process (sweeping the genie under the rug).

When the model decides to make a change to one of the controls (throttle, stick-pitch, or stick-roll), it sends control adjustment commands to the control inputs process of the STE and effects a motor movement in the model. The RPM component of ACT-R was designed to interface with a computer and implements device interfaces for a monitor, keyboard and mouse, but has no built in device interface for a stick or throttle. However, it is possible to extend the device interface class of RPM to handle new devices. For our purposes, it was only necessary to implement the virtual device interface for a stick and throttle since the model does not physically manipulate a stick or throttle and there is no need to model inputs from an actual physical device. Given the device interface classes and functions available, implementation of the virtual device interface for the stick and throttle was reasonably straightforward.

How do you handle time synchronization?

We don't. The model sends trigger press commands to the STE to initiate a trial. Immediately afterwards (20 msec later), the STE sends a "start-trial" signal via the VIT, which is detected by a Lisp process which then starts the model. At this point, they are both running. The STE runs only in real-time, and the model is restricted to running in real-time via the "real-time" parameter in ACT-R.

Occasionally, the trigger press that gets sent to the STE or the start trial signal that comes from the STE will get dropped (presumable by the socket polling mechanism) and the cognitive model and STE will become unsynchronized. This is currently the major outstanding interface problem for the combined system.

What didn't work? (and how did we overcome those challenges?)

Originally, the model ran slower than wall-clock time. This problem was caused by the fact that we needed a Lisp-based HUD and ACL 5.0.1 had poor graphics processing capabilities. We kluged time synchronization by creating a "jump-through-time" production that brought ACT-R time up to STE time whenever the model selected a new instrument to attend (about once every second). This was not a satisfactory solution, since it gave ACT-R the ability to travel through time and did not address the fact that the model only had a fraction of the attend-decide-act cycles that humans have. Eventually, we upgraded to ACL 6.1 (when ACT-R supported it), and got some invaluable assistance from Dan Bothell in implementing more efficient graphics code. Now the model runs faster than real-time, so we just turn on the "real-time" flag in RPM so that it runs at a realistic pace.

Naturally, we want to be able to do large batch runs for data collection and parameter optimization, but when we first implemented a batch run capability, the model would grind to a halt after about 20-25 trials. Eventually we figured out that this was caused by printing output of model activity to the Debug Window – an ACL issue. The buffer associated with the debug window in ACL accumulates content, which increasingly drains memory, and eventually bogs down the system since there is no mechanism for flushing or restricting the size of the buffer. The fix was very simple: we stopped printing to the Debug Window. Now we can run 1000's of trials without a memory issue.

References

Ball, J. T., Gluck, K. A., Krusmark, M. A., & Rodgers, S. M. (2003). Comparing three variants of a computational process model of basic aircraft maneuvering. *Proceedings of the 12th Conference on Behavior Representation in Modeling and Simulation*, 87-98. Institute for Simulation & Training.

Gluck, K. A., Ball, J. T., Krusmark, M. A., Rodgers, S. M., & Purtee, M. D. (2003). A computational process model of basic aircraft maneuvering. *Proceedings of the 5th International Conference on Cognitive Modeling*, 117-122. Universitats-Verlag Bamberg.

Schreiber, B. T., Lyon, D. R., Martin, E. L., & Confer, H. A. (2002). *Impact of prior flight experience on learning Predator UAV operator skills* (AFRL-HE-AZ-TR-2002-0026). Mesa, AZ: Air Force Research Laboratory, Warfighter Training Research Division.